Important Methods:

1. Search-Tree Node

For each node, we keep the State, a reference to the parent Node, the Operator that led to this node, number of Deaths, number of Kills, depth and the path cost and for the heuristics we also keep track of the expected deaths and expected kills. Upon creating the Node, we give a new state, a reference to the parent node, and the operator that was applied on the parent node and we calculate the path cost and depth for all nodes except the root node.

2. Matrix Problem

We have a generic **SearchProblem** class that is **abstract** and contains the **initial state** of the problem as well as the **list of operators** applicable for it. The **Matrix** class inherits from the **SearchProblem** class and it also contains instance variables related to the Matrix problem. It contains a 2D Array (grid) representing the problem, the max number of hostages to be carried (**maxCarry**), the **positions** of **Neo** and the **TelephoneBooth** as well as the **positions** of all **agents**, **hostages**, **pills**, **start pads** and **finish pads**. The **constructor** for the matrix problem takes a **String grid** that is generated by the **genGrid** function. Upon creating a matrix problem, we fill the **2D grid** with everything in the initialized grid string and set all variables in the class and initialize the **operators** also. In the **solve** method, we create a new **Matrix** problem and the **initialState** of the problem as well. For quick access of variables, we tend to use an

Object instead of the **String State** in each **Node** and that's why we created a **StateObject** class and used it for checking. We pass the initialNode to the chosen search procedure. The search procedure returns the Node that passes the **goalTest**. If a solution is found and we get a node back from the search procedure, we **backtrack** to find the **path from parent node to the goal node** and the **operators** used on this path as well as the **number** of **deaths**, **kills** and number of **expanded** nodes by this **search procedure**.

3. Main Implemented Functions:

a. **goalTest**: In the goal test we make sure that all hostages in the current state are either killed or rescued. Killed hostages are turned hostages and rescued hostages can be either alive or dead but not turned.

b. **search**: We have an **abstract** class **SearchProcedure** that does the search functionality. The abstract class contains three abstract methods which are **enqueue**, **dequeue**, **isEmpty** such that each sub-class should implement these methods based on the search procedure and the data structure used and how the search handles insertion and removal of nodes from the data structure. We have a **HashSet** to keep track of duplicate states so that nodes with duplicate states will not be inserted in the data structure. In the **search** method, while the **data structure** is not empty, we get the node and check if it **passes the goal test**. If it passes then the search is done and we reach a solution otherwise we check which of the **operators** can be

applied to the **state** in the current node. We apply the **valid** operators only and then calculate the state of the next time step after adjusting the hostages damage and add the output node after applying the operator to the data structure if its **new state** is **not a duplicate** and **neo is still alive**.

c. **genGrid:** First, we generate the grid dimensions randomly between 5 and 15 as well as the max number of hostages carried by Neo. Then we add all possible positions (x,y) to an arraylist and shuffle it so that the positions are not in order. Then we add all possible positions to a queue and dequeue a position from the front at each time we need a new position.
d. **isActionDoable:** Each operator has this function to check whether this action can be done or not before expanding the node and applying the time step. (There is an abstract class Operator which has 2 abstract methods which are **isActionDoable** and **apply** so that each operator inherits from Class Operator has these 2 functions customized for its use.)
e. **apply:** it applies the operator over the state of a node and returns the new state after the operator is applied.
f. **visualize:** It takes a state and visualizes the grid at such a state and the positions and state of all hostages, agents and pills.

4. Various Search Algorithms:

a) **BF**: The data structure used is a queue which follows the FIFO approach in inserting the nodes.

b) **DF**: The data structure used is a stack which follows the LIFO approach in inserting the nodes.

c) **UF**: The data structure used is a priority queue which inserts the nodes such that the node whose state has the minimum number of deaths should be inserted first. If there's a tie, then the node whose state has less number of kills is inserted first. That's the cost of each node **g(n)**.

d) **ID**: The data structure used is a stack which uses LIFO approach and it also considers the current level. If the max level is reached, no new nodes are added to the stack. For each node, a normal DF is used till the stack is empty. Then we increase the level by 1, insert the root node and run DF search with updated depth level.

e) **GR1**: The data structure used is a **priority queue** which inserts the nodes such that the node with **less** number of expected **deaths** shall be inserted first. That's the first heuristic **h1(nBF)**.

f) **GR2**: The data structure used is a **priority queue** which inserts the nodes such that the node with **less** number of expected **kills** shall be inserted first. That's the second heuristic **h2(n)**.

g) **AS1:** The data structure used is a priority queue which inserts the nodes such that the node with the minimum number of actual deaths and kills from the root **g(n)** and minimum number of expected deaths shall be inserted first. That's the number of **deaths** and **kills** from root (actual cost) **g(n)** and the **expected** number of **deaths** till a goal state is reached **h(n)**.

h) **AS2:** The data structure used is a priority queue which inserts the nodes such that the node with the minimum **number of actual deaths and kills** from the root **g(n)** and **minimum number of expected kills** shall be inserted first. That's the actual number of deaths and kills from root **g(n)** and the **expected** number of **kills** till a goal state is reached **h(n)**.

5. Heuristic Functions:

a. h1(n) Minimize the expected number of deaths:

We try to minimize the expected number of deaths as possible by calculating the number of hostages that will die if I try to go and carry them and drop them in the telephone booth by taking into consideration that Neo will take all untaken pill to increase the chance that the hostage will live and also considering if taking any of the pads will make it faster to reach the telephone booth and save the hostage. This process is done for each hostage so that if a hostage dies, then there is no way to make it live after considering the most optimistic scenario using all available pills and only rescuing this hostage and making use of pads if they will make the number of moves less. We use the manhattan distance to calculate the distances from neo to pills and the telephone booth, the hostage and the pads. This is an **admissible heuristic** because it will never overestimate the cost at any condition. The number of expected deaths will always be less than or equal to the actual cost because the only aim of this approach is to calculate the inevitable deaths that must happen even if I take all

available pills and use the shortest paths to save a certain hostage. The dead hostage can be turned and killed or dead while being carried.

b. h2(n) Minimize the expected number of kills:

In this heuristic, we try to calculate the number of expected kills such that the node with less number of expected kills should be expanded first. The idea here is to calculate how many hostages will be turned into agents and must be killed. These turned agents must be killed and we try to prioritize the nodes with less number of expected kills. We take into consideration taking all untaken pills as well as make sure of the shortest distance using any pads that can help reach the hostage without it being turned. The distance used is the manhattan distance from neo, to hostage, pills, pads and the telephone booth. This **heuristic is admissible** because it will never overestimate the number of kills because if after taking the pills and checking the pads, the hostage will inevitably die, then we have no option but to kill it.

6. Two Running Examples:

Example 1:

5,5;2;3,2;0,1;4,1;0,3;1,2,4,2,4,2,1,2,0,4,3,0,3,0,0,4;1,1,77,3,4,34

BF:

right,right,carry,left,left,left,up,up,carry,up,drop;0;0;2459

DF:

up,up,up,left,down,carry,up,drop,down,down,down,kill,up,up,up,left,down,down,down,down,right,right,up,up,up,up,right,takePill,down,down,down,down,left,up,up,up,up,left,down,down,down,down,left,up,up,up,up,right,down,down,down,down,right,up,up,up,up,right,down,down,down,kill,up,up,up,left,down,down,down,down,left,up,up,up,up;1;2;76

UC:

right,right,carry,up,up,up,left,takePill,down,down,down,left,left,up,up,carry,up,drop;0;0;76

ID:

up,up,right,down,down,down,right,up,carry,up,up,up,left,left,left,drop,kill;1;1;484

GR1:

right,right,carry,up,up,up,left,takePill,down,down,down,down,left,kill,left,up,up,up,carry,up,drop;0;1;110

GR2:

up,up,up,left,down,carry,up,drop,down,down,down,kill,up,up,up,left,down,down,down,down,right,right,up,up,up,up,right,takePill,left,down,down,down,down,left,up,up,up,up,left,down,down,down,down,right,up,up,up,up,left,down,down,down,fly,down,down,kill,up,up,left,down,down,down,down,left,up,up,up,up,left;1;2;140

AS1:

right,up,up,up,takePill,right,down,down,down,carry,up,up,up,fly,up,up,up,right,drop,down,carry,up,drop;0;0;96

AS2:

right,right,carry,up,up,up,left,takePill,down,down,down,left,left,up,up,carry,up,drop;0;0;76

Example 2:

5,5;2;0,4;3,4;3,1,1,1;2,3;3,0,0,1,0,1,3,0;4,2,54,4,0,85,1,0,43

BF:

left,left,left,left,down,carry,down,down,kill,right,right,down,carry,right,right,up,drop;1;2;10
8013

DF:

down,down,down,down,left,up,up,takePill,up,up,left,down,kill,up,left,down,down,kill,up,u
p,left,down,carry,up,right,down,down,down,down,kill,up,up,up,up,left,down,down,down,
down,right,kill,up,up,up,up,right,down,down,down,down,right,up,up,up,up,right,down,do
wn,down,drop;3;4;65

UC:

left,down,down,takePill,up,up,left,left,fly,down,carry,right,right,carry,right,right,up,drop,lef
t,up,left,left,left,up,carry,up,right,right,down,right,down,down,right,drop;0;0;1092

ID:

down,down,down,down,left,up,up,up,up,left,down,kill,up,left,down,down,left,down,kill,up
,up,carry,right,down,right,down,kill,right,right,drop;3;4;1755

GR1:

left,down,down,takePill,right,down,down,left,left,carry,up,kill,right,up,up,up,left,down,kill,
up,left,down,down,down,down,kill,up,up,up,up,right,down,down,down,down,right,up,up,
up,up,left,down,left,kill,up,left,down,down,down,down,right,up,up,right,up,up,right,down,
down,down,down,right,up,drop;3;4;154

GR2:

left,left,left,kill,fly,kill,down,carry,up,right,up,right,right,takePill,down,left,down,left,left,up,
up,up,carry,up,right,down,down,down,down,right,up,up,up,up,right,down,down,down,do
wn,right,up,drop,up,up,up,left,down,down,down,down,kill,up,up,up,up,right,down,down,
down;3;3;251

AS1:

left,down,down,takePill,down,down,left,left,left,carry,right,right,right,right,up,drop,down,l
eft,left,carry,up,up,left,left,up,carry,down,right,right,down,right,right,drop;0;0;737

AS2:

left,down,down,takePill,up,up,left,left,fly,down,carry,right,right,carry,right,right,up,drop,left,up,left,left,left,up,carry,up,right,right,down,right,down,down,right,drop;0;0;1092

7. Performance Comparison

Test Example:

5,5;3;1,3;4,0;0,1,3,2,4,3,2,4,0,4;3,4,3,0,4,2;1,4,1,2,1,2,1,4,0,3,1,0,1,0,0,3;4,4,45,3,3,12,0,2,88

BF:

RAM:1.8GB

CPU: 40%

Optimality: No

completeness: Yes

Number of expanded nodes:1382196

DF:

RAM:0.2GB

CPU: 25%

Optimality: No

completeness: No

Number of expanded nodes: 90

ID:

RAM:0.48GB

CPU: 30%

Optimality: No

completeness: No

Number of expanded nodes: 14477

UC:

RAM: 0.1GB

CPU: 20%

Optimality: Yes

completeness: Yes

Number of expanded nodes: 1631

GR1:

RAM: 0.17GB

CPU: 18%

Optimality: No

completeness: No

Number of expanded nodes: 338

GR2:

RAM: 0.11GB

CPU: 20%

Optimality: No

completeness: No

Number of expanded nodes: 237

AS1:

RAM: 0.15GB

CPU: 25%

Optimality: Yes

completeness: Yes

Number of expanded nodes: 1602

AS2:

RAM: 0.13GB

CPU: 20%

Optimality: Yes

completeness: Yes

Number of expanded nodes: 1631

Comparison:

RAM Usage:

The BFS algorithm is the one with the most RAM usage because it keeps expanding one level at a time which consumes a lot of memory. The uniform cost and heuristics algorithms are fast because they are not blind search and they try to reach the goal faster.

CPU Utilization:

The highest CPU utilization is achieved by the BFS because if the solution is deeper in the tree, it takes too long to reach it and expand all the nodes in the levels in-between. The second highest one is the iterative deepening because it keeps expanding nodes again and again for each max depth and it might go forever. The fastest algorithms are the greedy ones because they always tend to reach the goal faster using the heuristics. The uniform cost and admissible heuristics algorithms are a little bit higher than greedy because they guarantee optimality.

Number of expanded Nodes:

DFS expanded the least number of nodes 93 and also greedy expands very few number of nodes compared with other algorithms because greedy algorithms tries to find the best path and to expand the node which is expected to reach the goal faster.The uniform cost search and the admissible heuristic algorithms are close to each other but the admissible heuristic algorithms tend to expand less nodes because of the heuristic which tries to pick nodes closer to the goal. BFS expanded the most

number of nodes with huge difference between it and the one below it because it tries each operator at a certain level before moving to the next.